IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION PAPERS

OF

ANDREW MARK NIGHTINGALE

AND

ALISTAIR CRONE BRUCE

FOR

SOFTWARE AND HARDWARE SIMULATION

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates to the field of data processing systems. More particularly, this invention relates to the simulation of data processing systems including both a software component and a hardware component.

Description of the Prior Art

It is widely recognized that verification can take up to 70% of the development effort for a system on a chip (SoC) design. This includes software as well as hardware verification.

Most hardware verification is done by simulation, however, until recently, efficient software verification has had to wait for the hardware design to finish and be synthesized onto a field programmable gate array (FPGA). Then the software could be run on a prototype development board, but this wait inevitably increased the development time.

One solution is to simulate the software and the hardware parts together using commercial co-simulators. One notable benefit this approach is the quality of the software component is much improved as the co-verification has eliminated errors that would otherwise only be found during system integration.

IP blocks are self-contained designs for common functions that can be re-used in a SoC, saving time and effort for the main function. During their development, verification can be done in the same way as for a full SoC and with the same benefits. However, an IP block is intended for re-use and so its verification should also be re-useable. In fact, an IP block is only viable if it takes less effort to integrate into a system than it would to develop the block from scratch.

Under these circumstances it is to be expected that nearly all the integration effort will be in verification. It has been suggested that verification represents up to 90% of the effort to integrate an IP block into a system.

An IP block should therefore be considered as having three components: hardware, software *and* verification. Re-useable verification is an important part of an IP block as it enhances its value by reducing the system integration effort. In other words, the value of the hardware and software are the reason for using an IP block but the verification component makes their re-use possible.

From this, it follows that the verification supplied with an IP block should be aimed at the needs of integration, not just development. Because IP blocks tend to be used as supplied, with no changes apart from those required by integration, functional verification of the block is less important. That has already been done during its development. Rather, the verification should be designed to show that the rest of the system correctly supports the IP block and that its presence does not upset the other parts of the design. This is very different from the standalone verification often supplied to verify synthesis of a soft IP block. Currently, the verification component supplies a kit of verification parts that can be used to build a set of verification tests as part of a system validation plan.

The latest verification techniques make use of high-level verification languages (HVL) supported by tools such as Veristy's Specman Elite or Sysnopsis' Vera. These provide good links to hardware simulation environments. Their purpose built test vector generation and coverage analysis tools make verification much easier and more thorough.

These tools have now added facilities to allow for the simulation of embedded software running on a SoC design.

SUMMARY OF THE INVENTION

Viewed from one aspect the present invention provides a method of simulating a system having a software component and a hardware component, said method comprising the steps of:

(i) generating with a test controller a software stimulus for said software component and a hardware stimulus for said hardware component, said software stimulus and said hardware stimulus being associated so as to permit verification of correct interaction of said software component and said hardware component;

(ii) modelling operation of said software component in response to said software stimulus using a software simulator; and

(iii) modelling operation of said hardware component in response to said hardware stimulus using a hardware stimulator; wherein

(iv) said hardware simulator and said software simulator are linked to model interaction between said hardware component and said software component; and

(v) said software stimulus is passed to said software simulator by issuing a remote procedure call from said test controller to said software simulator.

The invention recognises that in many circumstances it is highly desirable to co-verify the correct operation and interaction of a software component and a hardware component of a design, such as, for example, the hardware design of a peripheral circuit and the associated peripheral driver software. Providing the ability to test in a simulated form the software component and the hardware component together enables more rapid product development since there is no longer a need to wait until test physical hardware becomes available before the interaction between the software component and the hardware component can be verified. Measures that reduce the time taken to develop new products are highly advantageous and give rise to strong competitive advantages. The ability to test the hardware component and the software component interacting together is facilitated by providing a remote procedure call mechanism whereby the test controller can pass a software stimulus to the software simulation. The remote procedure call mechanism for stimulating the software simulator allows the software component to be driven from the test controller which is already controlling the hardware simulation stimulation in a manner that avoids the need to develop specific test programs to be simulated by the software simulator for each design to be tested. Furthermore, providing the test controller with the ability to stimulate both the software simulator and the hardware simulator allows the test controller to closely monitor the interaction between the software component and the hardware component in a manner that strongly supports the ability to verify correct operation and interaction of the two components.

In preferred embodiments of the invention communication between the test controller and the software simulator is facilitated by the use of a shared memory via which call data specifying a software stimulus may be passed between the test controller and the software simulator. In this context, preferred embodiments utilise a start flag which may be set by the test controller within the shared memory to indicate to the software simulator that there is a pending software stimulus. The software simulator may correspondingly reset the start flag when it has completed modelling of the software stimulus.

The data written within the shared memory advantageously includes data identifying a software routine to be modelled within the software component and variable data to be used in responding to the software stimulus.

Whilst it will be appreciated that the above described technique can be utilised to test the correct operation of a wide variety of software components and hardware components that are associated in a manner such that their co-verification is desirable, the invention is particularly well suited to the co-verification of hardware peripheral devices and their associated software drivers. It will often be the case that these associated hardware and software components will be supplied together and will be unchanged between a wide variety of designs with the result that their rapid co-verification and removal from the design testing needs of a project is strongly desirable.

Preferred embodiments of the invention also provide for the monitoring of modelled signals at an interface with the hardware component that are generated in response to stimulation of the software component and the hardware component.

In this way, whilst the modelling and testing of the software component and the hardware component can effectively be bundled together into single process, the ability is maintained to monitor modelled signals at the hardware interface of the hardware component, so as, for example, to ensure that they obey predetermined rules.

The utilisation of a test controller to apply both software stimuli and hardware stimuli enables the coverage tools associated with such test controllers to be extended in use to ensure that an appropriate range of software stimuli have been applied in order to increase confidence in the testing that is performed.

As well as applying hardware stimuli to the hardware component, the test controller can operate to monitor the hardware to ensure that proper responses within the hardware are observed when corresponding software stimuli are applied. Thus, as an example, a software stimulus may be applied in the form of an instruction writing a particular value to a register within the hardware component and the hardware component can be monitored to ensure that physical signals corresponding to the value do appear within the appropriate register. The provision of a mechanism whereby a test controller can apply both software stimuli and hardware stimuli and monitor the resulting state changes to ensure that they match what is expected is strongly advantageous.

Viewed from another aspect the present invention provides apparatus for simulating a system having a software component and a hardware component, said apparatus comprising:

(i) a test controller operable to generate a software stimulus for said software component and a hardware stimulus for said hardware component, said software stimulus and said hardware stimulus being associated so as to permit verification of correct interaction of said software component and said hardware component;

(ii) a software simulator operable to model operation of said software component in response to said software stimulus; and

(iii) a hardware simulator operable to model operation of said hardware component in response to said hardware stimulus; wherein

(iv) said hardware simulator and said software simulator are linked to model interaction between said hardware component and said software component; and

(v) said software stimulus is passed to said software simulator by issuing a remote procedure call from said test controller to said software simulator.

Viewed from a further aspect the present invention provides a computer program product for controlling a computer to simulate a system having a software component and a hardware component, said computer program product comprising:

(i) test controller logic operable to generate a software stimulus for said software component and a hardware stimulus for said hardware component, said software stimulus and said hardware stimulus being associated so as to permit verification of correct interaction of said software component and said hardware component;

(ii) software simulator logic operable to model operation of said software component in response to said software stimulus; and

(iii) hardware simulator logic operable to model operation of said hardware component in response to said hardware stimulus; wherein

(iv) said hardware simulator logic and said software simulator logic are linked to model interaction between said hardware component and said software component; and

(v) said software stimulus is passed to said software simulator logic by issuing a remote procedure call from said test controller logic to said software simulator logic.

The above, and other objects, features and advantages of this invention will be apparent from the following detailed description of illustrative embodiments which is to be read in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 schematically illustrates a software component and a hardware component to be simulated together;

Figure 2 schematically illustrates a co-verification environment for co-verifying a software component and a hardware component;

Figure 3 schematically illustrates an alternative representation of a portion of the simulation environment of Figure 2;

Figure 4 schematically illustrates a further alternative view of the simulation environment of Figure 2;

Figure 5 illustrates an arrangement for allowing a test controller to issue remote procedure calls to a software simulator;

Figure 6 is a flow diagram illustrating the manner in which a remote procedure call corresponding to a software stimulus may be passed from the verification portion of a system to the simulation portion of a system;

Figure 7 is another representation of the simulation environment;

Figure 8 schematically represents a typical SoC structure with which the present techniques may be used; and

Figure 9 schematically illustrates a general purpose computer of the type that may be used to implement the above described techniques.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 illustrates a software component in the form of a smartcard driver 2 that serves to interact with a hardware component in the form of a smartcard interface 4. In practice both the smartcard driver 2 and the smartcard interface 4 may be provided as IP blocks by an organisation other than the user of those blocks in this instance. The smartcard driver 2 and the smartcard interface 4 will provide a small part of a larger SoC design. As well as providing the smartcard driver code and the data defining the smartcard interface circuit, such as integrated circuit layout defining data, the provider of these IP blocks may also provide a simulation of the smartcard interface 4 and the smartcard driver 2, or at least data that allows for their simulation. As illustrated, the smartcard driver 2 and the smartcard interface 4 can be considered together as a portion under test 6 which is supplied to a customer and which the customer wishes to test within their own design. In order to test the portion under test 6, the customer will apply software stimuli to the smartcard driver 2 in the form of software instructions at the driver API. Similarly, in order to test the smartcard interface 4, the customer will provide appropriate test stimuli that can be supplied to the model of the smartcard interface.

In practice, the software stimuli applied to the smartcard driver 2 may represent calls from an application program or calls from an associated operating system within the system as a whole that is being developed. The hardware stimuli may represent real physical signals that can be applied to the smartcard interface 4, such as signals that would result from insertion of a physical smartcard into such an interface in use, interrupt signals and other external signals.

It will be appreciated that by providing the portion under test 6 in a form that may be simulated as a whole, the customer need not wait until physical hardware is available before they can verify the correct operation of the smartcard driver 2 with the smartcard interface 4 within their overall system. There may well be large amounts of other testing that still needs to be performed, but the task of verifying the correct interaction of the smartcard driver 2 and the smartcard interface 4 can be pushed earlier into the development process in a manner that speeds overall development time.

Figure 2 schematically illustrates a test environment. A software driver 8 and a corresponding hardware peripheral device 10, represented by RTL data that may be used to control a simulator, are both provided. A bus interface module 12 handles the mapping of the response to software instruction execution into appropriate physical signals upon the peripheral bus 14 for passing to the hardware peripheral device 10. The bus interface module 12 will typically be a re-usable entity within the system that can be employed with a wide variety of software being simulated and hardware components that are to be driven and responded to.

The driver software 8 is simulated within an instruction set simulator 16. Software stimuli are passed to the software driver 8 via its API.

The hardware peripheral device 10 may be simulated using known hardware simulation techniques. The RTL data defining the hardware peripheral device 10 defines the hardware device to be simulated to the simulation software.

A test controller 18 is shown surrounding the instruction set simulator 16 and the hardware peripheral device 10. This test controller 18 may have the form of a verification tool or validation test bench. As well as providing appropriate hardware stimuli, such as hardware test vectors, such a test controller 18 may also provide software stimuli to the instruction set simulator 16. The hardware stimuli may be provided in the form of an external verification component 20, such as may for example model the smartcard hardware stimuli referred to in Figure 1. The software stimuli may be provided by the test controller as a test scenario 22 which serves to generate appropriate API calls to the software driver 8 as well as coordinating these calls with associated hardware stimuli. A conformance and coverage element 24 within the test controller 18 serves to monitor the stimuli applied to and responses of both the software driver 8 and the hardware peripheral device 10 to ensure that these conform with predetermined rules and to monitor the coverage of the range of potential stimuli that may be used in order to ensure an appropriate broad range of stimuli are applied.

Figure 3 shows the proposed verification structure for an IP block. The symmetry emphasizes the equal status of the hardware and software components. Notice that the communication between the software and hardware components is considered to be part of the IP block and as such is not directly driven by the testbench. Instead, it is checked by an interconnection-fabric protocol checker.

The two interfaces to the IP block that are driven by the testbench are the software interface of the driver API and the non-fabric hardware interface to other parts of the system.

A test scenario manager generates stimuli and collects responses from these two interfaces and uses scoreboarding to create self-checking, system level tests.

Both the hardware and software components of the IP block are checked for coverage. This is an important part of the methodology as it allows the user of the IP verification to quantify the coverage of the tests. This, in turn, means that the system validation can be checked for its coverage of a given IP block.

The internal state of the hardware and software components is monitored to measure coverage. The test scenarios can also use the state to verify correct internal behavior during the tests.

The hardware interface is accessed from the test environment via a bus functional model (BFM). This allows the tests to be written at a higher level of modeling, leaving the BFM to add the fine details.

The software interface is accessed from the test environment through the driver's API. This is in contrast to other methodologies where test programs have to be written and run on the co-simulation model in order to exercise the driver.

All the verification components have documented verification interfaces that allow them to be re-used in system tests without needing to know their internal working. For example, the system validation plan may call for a particular test to be performed with a FIFO both full and empty. The verification API and the coverage analysis supplied with the IP block give the appropriate system level facilities to implement this.

An important part of this methodology is the ability to call driver routines from the verification environment. This has been achieved by setting up a remote procedure calling mechanism between the software running on an ARM co-verification model (in Mentor Seamless) and a testbench running on a verification simulator (from Verisity Specman).

Figure 4 schematically illustrates an example of the present technique implemented using the Specman and Seamless systems previously discussed. In this figure the test controller is provided by the Specman component 26. The hardware simulator 28 and the software simulator 30 are both provided within the Seamless system. The Seamless system provides a programming interface that may be used by the Specman system 26 to pass appropriate software stimuli to the software simulator 30.

A kernal 32 within the Seamless system provides memory management such that the memory accessible to both the software components and the hardware components may be modelled in a unified manner with appropriate parameters associated with different portions and with the software simulator and hardware similator reacting to accesses to different memory locations in different ways depending upon what is being simulated and what is represented by that memory.

The link provided between Specman and Seamless gives the ability to read from and write to global variables in the software simulation. It is also possible for methods in the testbench to wait for software variables to change value. This is enough to implement a single threaded remote procedure call (RPC) from an ***e*** language testbench to software routines running on the simulator.

A simple client-server stub implementation is used, see Figure 5. On the client side, an ***e*** language struct is declared that encapsulates the client RPC functionality. The struct's init routine is extended to establish the connection with Seamless while the client stubs are declared as time consuming methods of the struct. Each client stub marshals the parameters and transfers them, along with a routine identifier, to a set of global variables in the software simulation. A further global variable is used to signal the start of the software routine. The client stub then waits for the software routine to signal its completion by resetting the start global variable.

The server side is implemented as a simple polling loop. Just before the loop is entered a global variable is set to signal that the server is ready. This allows the testbench to synchronize with the SoC co-simulation so that the hardware reset can complete along with any bootstrap activity.

The polling loop checks for the "start" global variable to be set (by the client stub) indicating that a routine is to be called. When it is set, a switch statement selects the routine to call with the parameters already cached in global variables. When the routine completes the start global variable is reset and the polling loop resumes.

The mapping between ***e*** and the global variables has to be considered. The default transfer size is 32 bits so for types that fit into a single 32-bit word, such as int, bool and char, the default mapping is correct. The connection offers the ability to change the endianess if required.

Multi-word types are transferred as a list of bit and then unpacked into an ***e*** structure. In these cases the ***e*** definitions must be padded to the correct number of bits to match the alignment in the software simulation.

API Routines that have pointer parameters are handled specially. Matching data areas are set up on the client and server. The client transfers the server data across, modifies it as required and then transfers it back again. The routine can thus be called from the server stub with the pointer to the server data area.

Very often, a driver API makes use of call back routine parameters. This implies a call from the software simulation back into the into the verification code. These are handled by passing the address of a client stub routine on the server. This then uses the same RPC technique to call a routine via a server stub in the verification code.

Once the driver is accessible from the high-level verification language it is possible to write the sort of randomized test that has proven so useful in the hardware context. For example, it is now easy to generate a set of random parameter values for a routine and check its result. Such random parameter values within constrained valid ranges may be automatically generated for the software stimuli by the test coordinator.

Both multi-word parameters and HVL facilities to randomly exercise a driver routine may be used. Normally the results would be checked against the expected values to form a self-checking test.

Figure 6 is a flow diagram schematically illustrating the processing performed within the test controller (verification) portion and the simulation portion when using the remote procedure call technique to pass to a software stimulus.

At step 34, the verification software waits until a test scenario being used stimulates a client stub. When a client stub has been stimulated, then step 36 serves to martial the required variables that need to be passed to the software driver concerned. At step 38 the variables to be passed are written to a shared memory 40. Once this set up is complete, step 42 serves to set a start flag within the shared memory 40 to indicate that a software stimulus for the software simulator is pending and needs to be serviced. Processing within the test controller continues at step 44 waiting for the start flag to be reset before returning to step 34.

Within the simulation software a polling loop serves to monitor the start flag using step 26 to detect when it is set. When the start flag is set, step 48 reads the associated variables from within the shared memory 40 and then the software stimulus specified is simulated at step 50 using the instruction set simulator and associated driver software. When this is complete, step 52 resets the start flag and processing on the simulation side returns to step 46.

The co-simulation environment may be provided by Mentor Seamless CVE, which contains an ARM co-verification model, and by Mentor ModelSim VHDL simulator. Verisity Specman Elite may provide the verification environment (test controller) with links to both the HDL simulation and the software running on the ARM co-verification model. The link to the software is available in Specman Elite version 3.3 and Seamless CVE version 4.0. It is a key enabling technology for this methodology.

The mapping of the verification structure onto these tools is shown in Figure 7. Seamless supports the simulators to animate the software and hardware components of the IP block. The software is run on the cycle accurate instruction set simulator (an ARM co-verification model) whilst the ModelSim VHDL simulator animates the hardware component. Communication between the two simulators is mediated by the Seamless CVE co-simulation kernel and a bus interface model.

The verification components and test manager are implemented in the *e* language supported by Specman Elite.

A SmartCard Interface PrimeCell may be incorporated into a typical SoC structure as shown in Figure 8. This was based on the AHB/APB VHDL testbench supplied as part of MicroPack 2.0[1]. For the SmartCard Interface example, a smartcard bus functional model may be written to give verification access to the hardware API. It is advantageous that an object-oriented language, such as ***e***, is used for implementation as it allows the verification components to be safely modified to meet the needs of a particular test or system. For example, a smartcard may need to support a specialized encryption algorithm. With object-oriented techniques (*extend* in the case of ***e***) this can be easily layered on top of the basic functionality provided with the original verification component.

With the verification structure in place, a number of test scenarios may be implemented to use the new methodology. These may be coded as extensions to the appropriate verification components. A good example is the initial card activation sequence, taking the card as far as the Answer to Reset (ATR) response. This demonstrates the ability to use the full driver API from the verification environment.

First the driver is initialized and then the test waits for the driver to callback when the card is inserted. Meanwhile, the card verification component has been configured (using constraints) to insert at a certain time, to send an ATR after coming out of reset, and finally to remove the card at some later time. The ATR is randomly generated using the constrained generation facilities of the ***e*** language.

When the driver calls back to indicate that the card has been inserted, the driver API is called to activate the card. Then, the test waits for the ATR to complete (another callback) when it can compare the ATR sent by the card with the ATR received by the driver.

A variation of this test, where the card is removed part way through the ATR, is easily implemented by just changing the card component's removal time.

Another example test activates the card, writes to the card and then reconciles the data that arrives at the card with the data sent from the testbench.

These tests demonstrate that this methodology provides full end-to-end checking of an IP block from the software driver's API through to the pin interface of the hardware.

Figure 9 schematically illustrates a general purpose computer 200 of the type that may be used to implement the above described techniques. The general purpose computer 200 includes a central processing unit 202, a random access memory 204, a read only memory 206, a network interface card 208, a hard disk drive 210, a display driver 212 and monitor 214 and a user input/output circuit 216 with a keyboard 218 and mouse 220 all connected via a common bus 222. In operation the central processing unit 202 will execute computer program instructions that may be stored in one or more of the random access memory 204, the read only memory 206 and the hard disk drive 210 or dynamically downloaded via the network interface card 208. The results of the processing performed may be displayed to a user via the display driver 212 and the monitor 214. User inputs for controlling the operation of the general purpose computer 200 may be received via the user input output circuit 216 from the keyboard 218 or the mouse 220. It will be appreciated that the computer program could be written in a variety of different computer languages. The computer program may be stored and distributed on a recording medium or dynamically downloaded to the general purpose computer 200. When operating under control of an appropriate computer program, the general purpose computer 200 can perform the above described techniques and can be considered to form an apparatus for performing the above described technique. The architecture of the general purpose computer 200 could vary considerably and Figure 9 is only one example, e.g. a server may not have a screen and a mouse or keyboard.

Although illustrative embodiments of the invention have been described in detail herein with reference to the accompanying drawings, it is to be understood that the invention is not limited to those precise embodiments, and that various changes and modifications can be effected therein by one skilled in the art without departing from the scope and spirit of the invention as defined by the appended claims.